Problem Solving with Java

Session 4

Problem Set #2

- Implement 4 kinds of uninformed search:
  - DFS, BFS, DLS, IDS
- Demonstrate that it works
- Test it.
- Do some analysis
- Preview of PS3
  - More testing of 4 search algorithms
  - Extend code to add A* search
  - Etc.

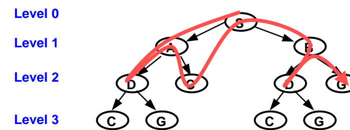
What we expect

Implementation should be well written!

- Good use of data and procedural abstraction.
  - Define abstract data types rather than using built in types.
  - Don't violate abstraction barriers.
- Good use of classes
  - Abstract data types.
  - Appropriate use of subclassing
  - Appropriate use of interfaces
- Good testing with Junit
- Generalize generalize generalize…

Depth-Limited Search

Same as Depth-First Search except that after the depth limit has been reached it doesn't go any deeper.



Compare BFS & DFS

- Depth First Search – much slower than BFS
- Breadth First Search – uses much more space than DFS
- Breadth First finds shortest path – not DFS
- Breadth First search guaranteed to find path

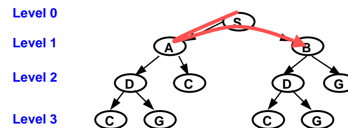
	Time	Space	Optimal	Find Path
DFS		☺		
BFS	☺		☺	☺

Iterative Deepening (IDS)

Idea:

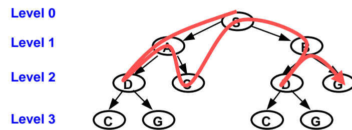
• Explore tree in breadth-first order, using depth-first search.

➔ Search tree to depth 1, ….



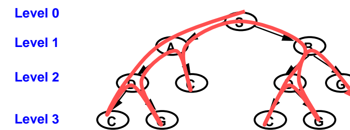
Iterative Deepening (IDS)

Idea:
• Explore tree in breadth-first order, using depth-first search.
➔ Search tree to depth 1, then 2, ….



Iterative Deepening (IDS)

Idea:
• Explore tree in breadth-first order, using depth-first search.
➔ Search tree to depth 1, then 2, then 3….



Summary

- Complexity analysis shows that breadth-first is preferred in terms of optimality and time, while depth-first is preferred in terms of space.
- Iterative deepening draws the best from depth-first and breadth-first search.

Abstract classes and interfaces

An abstract class is partially implemented

Some declared methods are not implemented
The class cannot be instantiated
Subclasses inherit implementations, fields
A concrete subclass must implement all methods

An interface is not at all implemented

Methods may be declared
No fields, no constructor

These are useful when you want certain functionality, but do not wish to impose an implementation

Generally:

Abstract class for closely related classes
Interface for functionality not dependent on the class

Abstract Classes

```
abstract class Shape {
  Color color; // color of shape.
  void setColor(Color newColor) {
    // method to change the color of the shape
    color = newColor; // change value of instance variable
    redraw(); // redraw shape, which will appear in new color
  }
  abstract void redraw();
    // abstract method -- must be defined in
    // concrete subclasses . . .
    // more instance variables and methods
} // end of class Shape
```

Java interfaces

Java interfaces are a collection of method signatures

No implementation
Like a class with no method bodies (+ no constructor)

Great for creating a declared type with a weak specification

Eliminates needless dependences on actual (runtime) type of an object
Makes code much easier to reuse when new,
enhanced types are introduced

Subclassing and dispatch recap

```
class Account { ...
  boolean check(Transaction t) {
    return (balance + t.amount) >= 0;
  }
  void post(Transaction t) {
    if (check(t)) { ... }
  }
}

class OverdraftAccount extends Account
{ ...
  boolean check(Transaction t) {
    return (balance + t.amount) >= creditLimit;
  }
}

Account warbAcct = new OverdraftAccount(...);
Transaction t2 = new Transaction(...);
warbAcct.post(t2); // which check() does post() call?
```

Dispatching

Which version of check is called by post?

- When a method is called on an object, check the object's actual (run-time) type:
- OverdraftAccount
- Search there first for the method: not found
- If not found, search superclass: Account
- Found! So we call Account.post
- Which then calls check on warbAcct
- Exact same process again – search actual type first:
- OverdraftAccount
- OverdraftAccount.check found!

Role of declared and actual types

declared (compile-time) type

- Determines what *specification* the client (and compiler) relies on an object to satisfy
- Either a class or an interface
- A single object can be viewed as different compiletime types in different parts of a program

actual (run-time) type

- Keeps track of what *implementation* the object has
- This is always a class, never an interface
- The run-time type of an object never changes

Role of declared and actual types

instanceof tests whether an object is compatible with a particular declared (compile-time) type

```
Object o1 = Arrays.asList("happy","list","of","strings");
Object o2 = "lonely string";
System.out.println(o1 instanceof List); // true
System.out.println(o1 instanceof Collection); // true
System.out.println(o1 instanceof Comparable); // false
System.out.println(o1 instanceof String); // false
System.out.println(o2 instanceof List); // false
System.out.println(o2 instanceof Collection); // false
System.out.println(o2 instanceof Comparable); // true
System.out.println(o2 instanceof String); // true
```

Point example

```
class Point {
  private double x, y;
  public Point(double x, double y)
  { this.x=x; this.y=y; }
  public double x() { return x; }
  public double y() { return y; }
  public double r() { return Math.sqrt(x*x+y*y); }
  public double theta() { return Math.atan2(y,x); }
}
```

Adding caching

```
class CachingPoint {
  private double x, y;
  public CachingPoint(double x, double y)
  { this.x=x; this.y=y; }
  public double x() { return x; } // same as Point
  public double y() { return y; } // up to here
  private double r, theta;
  private boolean polarized;
  private void polarize() {
    if (!polarized) {
      r = Math.sqrt(x*x+y*y);
      theta = Math.atan2(y,x);
      polarized = true;
    }
  }
  public double r() { polarize(); return r; }
  public double theta() { polarize(); return theta; }
}
```

Reducing impact of extension

```
class CachingPoint extends Point {
  private double r, theta;
  private boolean polarized;
  public CachingPoint(double x, double y) {
    super(x,y);
  }
  private void polarize() {
    if (!polarized) {
      double x = x();
      double y = y();
      r = Math.sqrt(x*x+y*y);
      theta = Math.atan2(y,x);
      polarized = true;
    }
  }
  public double r() { polarize(); return r; }
  public double theta() { polarize(); return theta; }
}
```

Reducing impact of extension

```
class Graphics {
  void drawLine(Point p1, Point p2) { ...
  }
  ...
}
...
screen.drawLine(new CachingPoint(5,10),
new CachingPoint(10,5));
```

drawLine method doesn't need to be touched

Interfaces and specifications

```
public interface Matrix {
  int get(int row, int col);
  void set(int row, int col, int val);
  int rows();
  int cols();
}
```

Interfaces and specifications

```
public interface Matrix {
  // specfield width : integer
  // specfield height : integer
  // specfield cells : map from integer pair to integer
  // returns: cells[<row,col>] if it exists, otherwise 0
  int get(int row, int col);

  // modifies: this.cells
  // effects: cells[<row,col>] = val
  void set(int row, int col, int val);

  // returns: height
  int rows();

  // returns: width
  int cols();
}
```

Flat array implementation

```
public class OneDimArrayMatrix implements Matrix
{
  private int[] data;
  private int width;
  public OneDimArrayMatrix(int r, int c) {
    data = new int[r*c];
    width = c;
  }
  public int get(int r, int c)
  { return data[r*width+c]; }
  public void set(int r, int c, int v)
  { data[r*width+c] = v; }
  public int rows() { return data.length/width; }
  public int cols() { return width; }
}
```

2D array implementation

```
public class TwoDimArrayMatrix implements Matrix
{
  private int[][] data;

  public TwoDimArrayMatrix(int r, int c)
  { data = new int[r][c]; }

  public int get(int r, int c)
  { return data[r][c]; }

  public void set(int r, int c, int v)
  { data[r][c] = v; }

  public int rows() { return data.length; }
  public int cols() { return data[0].length; }
}
```

Why interfaces instead of classes

Java design decisions:

- A class has exactly one superclass
- A class may implement multiple interfaces
- An interface may extend multiple interfaces

Observation:

- multiple superclasses are difficult to use and to implement multiple interfaces, single superclass gets most of benefit

Potential benefits of subclassing

Implementation reuse

- Implementations need not repeat unchanged fields and methods – instead reuse from the superclass

Specification reuse

- Clients who understand the superclass specification need only study novel parts of subclass

Ability to substitute new implementations

- Clients may not need to change their code when new subclasses are developed, if they only handle